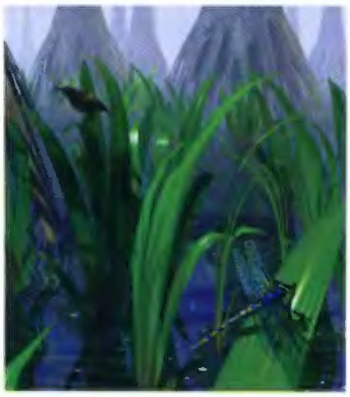


RENDERMANÍA

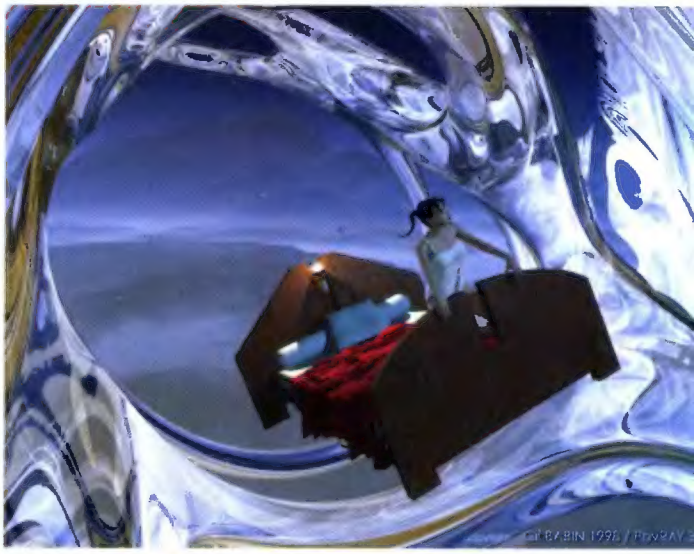
Taller Virtual Nuevas sentencias de programación (III)

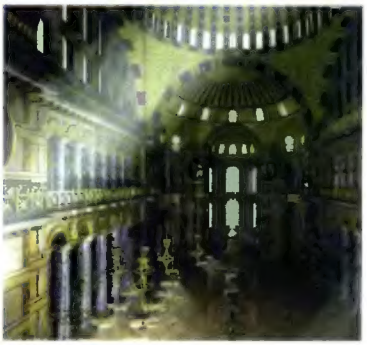
Cómo... El futuro de la animación con POV

En el CD Las mejores escenas de la Internet Raytracing Competition

El foro del lector Animaciones de gran tamaño

pcmanía





José Manuel Muñoz

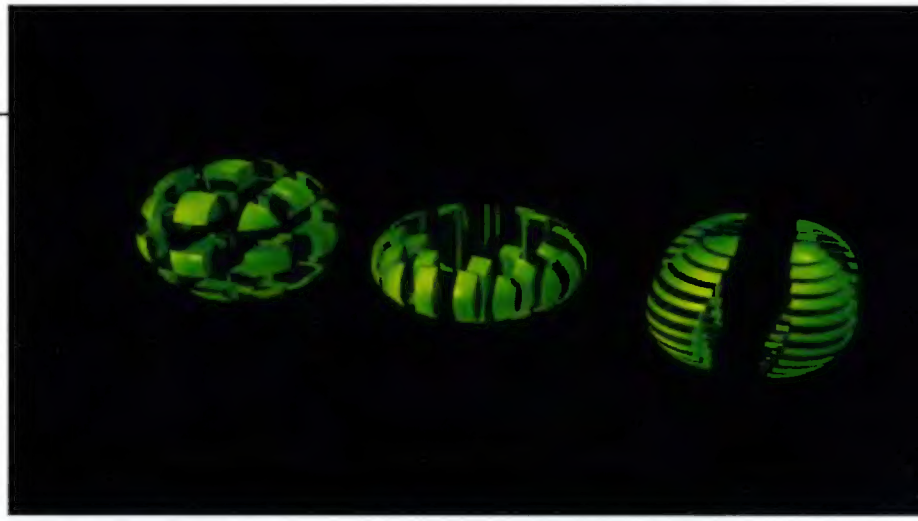
Los ejemplos y los ficheros de El Foro los encontraréis en el directorio PCMANIA/RENDER71 del CD-Rom.

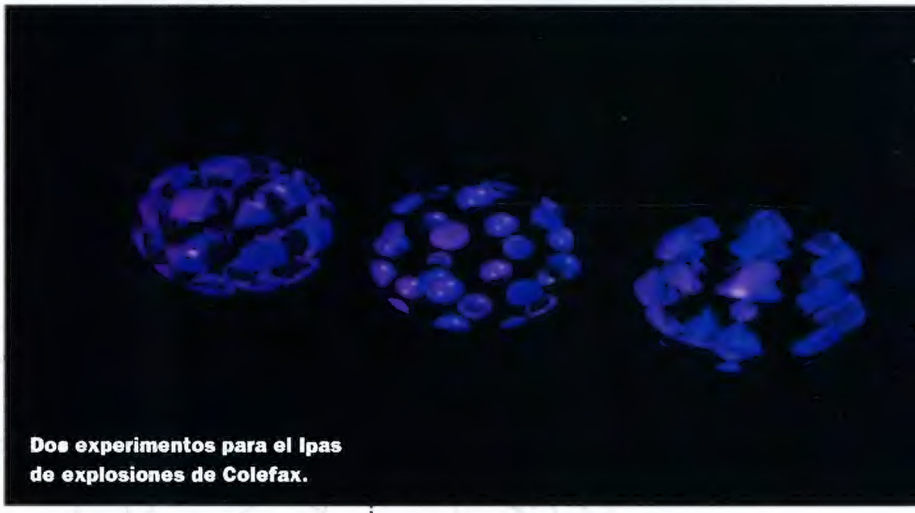
Sentencias de programación (parte III)

En esta ocasión estudiaremos dos de las sentencias de "programación" que han sido incluidas en la nueva versión de «POV». Se trata de #local y #macro, que van a suponer una revolución en la creación de escenas en general y en la programación de pov-ipas en particular. Con estas dos nuevas sentencias, el lenguaje escénico de POV pasa a ser un verdadero lenguaje de programación con el que podremos hacer cosas que antes sólo eran posibles con la utilización de utilidades externas.

Nuevamente queremos insistir en la importancia de las llamadas sentencias de programación. Las sentencias convencionales de construcción de objetos de «POV» nos permitirán crear modelos de gran complejidad, pero estas sentencias por sí solas no nos proporcionarán ninguna ventaja con respecto a los modeladores visuales (basados en la edición con ventanas de

trabajo). Es más, con «POV» es muy difícil –si no imposible– crear ciertos modelos como los que pueden ser construidos con modeladores como «Rhino», «Spatch», «MAX» y cualquier otro programa que utilice splines. Por esta razón, muchos usuarios de »POV» exportan a dicho programa modelos construidos desde otros programas y utilizan el lenguaje escénico de este raytracer para construir la escena situando los objetos, las luces y la cámara. Obrando así no sólo podremos aprovecharnos de la magnífica calidad del render de «POV», sino también de las posibilidades que ofrecen sus sentencias de "programación". Utilizando estas sentencias podremos manipular los modelos y generar escenas impresionantes cuya creación resultaría problemática o imposible desde un modelador visual. (Nota: Quizá «MAX» sea una excepción a esto. Parece que este programa dispone de un lenguaje propio muy potente y quizá con él podrían crearse escenas comparables a las del número 17 de Rendermanía).





Dos experimentos para el Ipas de explosiones de Colefax.

#Declare y #Local

En el número 17 de Rendermanía aprendimos a utilizar la sentencia #declare para definir objetos que más tarde podían ser referenciados y empleados cuantas veces fuese necesario. Vimos, además, cómo #declare podía ser utilizada para crear variables o cambiar el valor de éstas, siendo esto la base de toda la parte de "programación" que tiene el lenguaje escénico de «POV» (puesto que estas variables se emplean más tarde para estudiar las condiciones de sentencias como #if o #switch, o para tratar los contadores de #while). A continuación vamos a presentar una nueva sentencia incluida en la nueva versión 3.1 de «POV». Se trata de #local, cuyo uso y funcionamiento es prácticamente idéntico al de #declare. Por ejemplo, utilizando #declare...

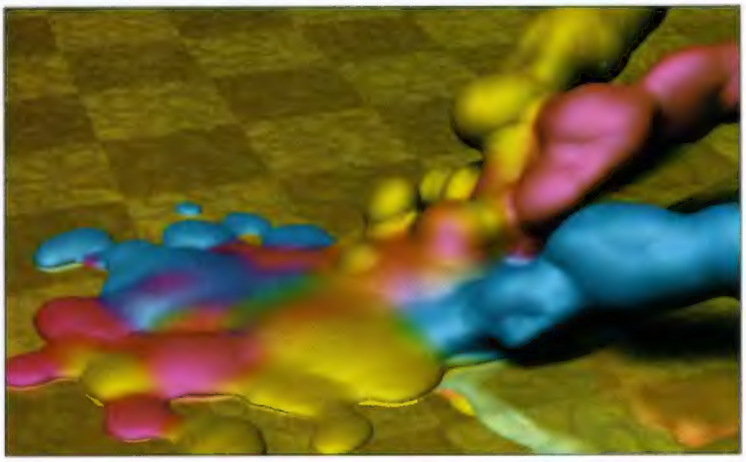
```
#declare xa=5;
#declare xa=xa+1;
```

la primera sentencia asignará a la variable xa el valor 5 y la segunda dejará a xa con el valor 6. Pues bien, las dos siguientes líneas harán exactamente lo mismo utilizando #local:

```
#local xa=5;
#local xa=xa+1;
```

Puesto que #local también se emplea de la misma forma que #declare, es decir, para asignar un objeto a un identificador, el lector estará empezando a preguntarse para qué se ha incluido a esta sentencia en el lenguaje de «POV". La respuesta está en que emplearemos #local para utilizar variables locales y #declare para emplear variables globales. (Sin duda, con esta última frase los aficionados a la programación creerán tenerlo todo claro, pero el caso es que las variables locales de «POV» funcionan de una manera algo diferente a como lo hacen las de C, de manera que será mejor que no os saltéis las explicaciones siguientes). Aquí, como en el lenguaje C, las palabras global y local se refieren al ámbito en el cual la variable será reconocida (por el compilador) y podrá ser utilizada. Las variables globales, una vez creadas, pueden ser referenciadas y empleadas a lo largo de cualquier parte del programa. Las va-

Spray es otro pov-ipas de Colefax.



riables locales, en cambio, solamente son conocidas dentro del bloque en el cual han sido definidas. Si se intenta referenciar a una variable local desde fuera del bloque en el que ha sido creada, entonces el programa dará un error indicando que esa variable no existe. Esto no incluye el caso de los bloques de código inscritos dentro del bloque donde se ha definido la variable local, como sucede en el caso de las macros (las cuales serán explicadas más adelante).

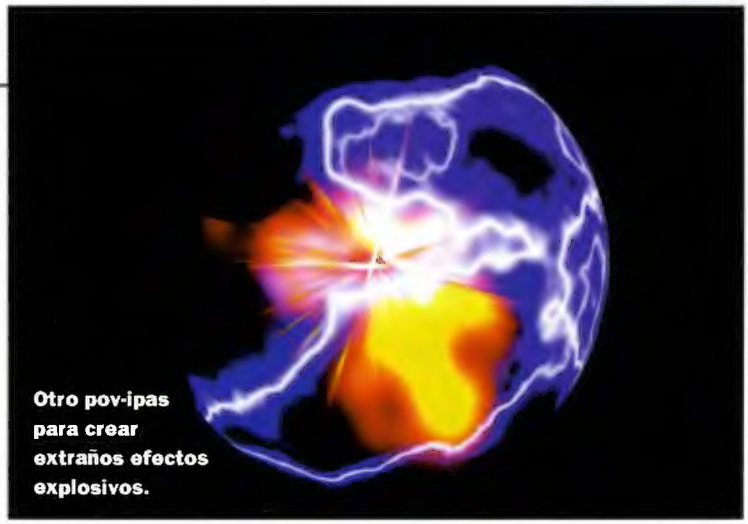
A continuación intentaremos explicar las reglas de alcance o ámbito por las que se rigen las variables locales. Para empezar, ¿a qué nos referimos con la palabra bloque? Bien, en el lenguaje C (con el que «POV» cada vez tiene más semejanzas) un bloque es cualquier extensión de código delimitada por llaves. En C un bloque puede ser el código inscrito dentro de las llaves de una función o el delimitado por las llaves de un if. Esto significa que, como podemos escribir sentencias if dentro de las funciones, puede haber bloques dentro de bloques. Supongamos que creamos una variable local dentro de un bloque C que está incluido dentro de un bloque B que se encuentra, a su vez, incluido dentro de un bloque A. Pues bien, la variable local definida dentro de C sólo será conocida por este bloque y no por B ni por A. (Se dice entonces que la variable es local al bloque C). Ahora supongamos que definimos a esta variable local dentro del bloque B en lugar de hacerlo en el C. En este caso la variable no será conocida por A pero sí por B y por C. ¿Por qué por C? Pues porque el bloque C está incluido dentro del bloque B y forma parte del mismo. Diciéndolo de otra manera: el bloque C se encuentra dentro del ámbito o alcance del bloque B. Esto significa que cualquier variable local de B es una variable local de C mientras que lo contrario no es cierto.

Estas mismas reglas de funcionamiento de ámbitos, tan bien conocidas por los programadores de C, se aplican del mismo modo a las variables locales de «POV» cuando trabajamos con macros, como veremos después. Pero lo dicho no rige cuando se están utilizando ficheros incluidos. Supongamos que tenemos un archivo C que es leído (con #include) por un fichero B que a su vez es leído por un fichero A. En este caso una variable local definida en el fichero B no sería reconocida dentro del fichero C ni dentro del A. En otras palabras, un fichero leído por otro no se considera como parte del bloque que forma el fichero lector.

Al llegar aquí es probable que los lectores no amantes de la programación se sientan algo liados. Enseguida pondremos algunos ejemplos pero antes reseñaremos dos puntos importantes. El primero es que, como recordaréis, en «POV» los bloques de código no se delimitan con llaves sino, como hemos visto en los casos de #if, #while y #switch, con la sentencia #end (que cierra el bloque de código abierto con la sentencia #if, #while o #switch que hayamos usado). El segundo punto, mucho más importante, estriba en que

las variables locales de «POV» no son locales a estos bloques de código, sino a ficheros y a macros.

¿Cómo? ¿Que aún no habéis entendido nada? Vale, vamos a poner un ejemplo práctico. Supongamos que tenemos un fichero llamado batalla.pov con el que pretendemos generar una escena donde se ilustra una de nuestras típicas batallas entre orcos y humanos. A este archivo podemos llamarlo el fichero principal del proyecto o principal a secas. Contendrá las definiciones de la cámara y las de las luces, y probablemente también una serie de bucles #while para situar las formaciones de contendientes en el pov-espacio. Como sabéis, por criterios de claridad y funcionalidad, es raro que un proyecto lo suficientemente complejo (como lo sería el del ejemplo) conste de un único fichero. En este ejemplo supondremos que batalla.pov tendrá en su código un par de sentencias #include para cargar a los archivos humano.inc y orco.inc. Si la escena muestra a dos ejércitos puestos frente a frente, parte del código de batalla.pov podría consistir en dos bloques bien diferenciados, constando cada uno de ellos de un doble bucle #while para crear una formación rectangular de guerreros. Dentro del centro de cada uno de estos bloques habría una sentencia #include de manera que, con cada pasada del bucle, se invocase a humano.inc o a orco.inc (dependiendo del bloque) para crear un modelo. Por supuesto, la función de humano.inc sería la de crear un humano en la posición espacial indicada por las variables del bucle y orco.inc realizaría la misma tarea para los orcos. Pues bien, imaginad que escribimos la siguiente línea en algún lugar dentro de batalla.pov, antes de las líneas #include a humano.inc y orco.inc:



Otro pov-ipas para crear extraños efectos explosivos.

```
#declare godel=25;
```

Como ha sido definida utilizando #declare, godel es una variable global y podrá ser invocada y alterada dentro de humano.inc y orco.inc. En caso de que la misma línea hubiese sido escrita dentro de, por ejemplo, humano.inc y no en batalla.pov, entonces la variable godel también seguiría siendo global (al haber sido definida con #declare) y podría ser referenciada desde batalla.pov, en cualquier línea posterior a la sentencia #include "humano.inc". En definitiva, una variable global puede ser referenciada y tratada desde cualquier fichero de un proyecto, una vez que ha sido definida.

Ahora supongamos que escribimos la siguiente línea dentro de humano.inc:

```
#local escher=26;
```

Esto declara a escher como una variable local al fichero humano.inc. Esto quiere decir que esta variable podrá ser tratada desde cualquier punto de humano.inc (posterior al de su declaración, claro está) pero no deberemos invocarla desde batalla.pov, so pena de error de compilación. En otras palabras, escher se convertirá en una variable local a humano.inc y, si escribiésemos en batalla.pov u orco.inc una línea como "#local escher=escher+1;" o como "#declare escher=escher+1;", entonces «POV» nos daría un mensaje de error. (Nota: recordad que cuando «POV» halla una nueva variable, la declaración no debe ser algo como "#declare variable=variable+1;", pues entonces «POV» interpretará que la asignación del valor depende de una variable a la que aún no se ha asignado valor alguno, es decir, que no existe).

Ahora veamos otro caso. Supongamos que humano.inc tiene, a su vez, en alguna parte de su código una sentencia #include que carga el fichero humanda.inc y que dentro de este último fichero existe la siguiente línea:

```
#local bach=27;
```

Pues bien, bach solo podrá ser utilizada desde humanda.inc y no será conocida por humano.inc ni por batalla.pov.

Nota importante sobre el punto y coma

Antes de seguir haremos un inciso para comentar que el símbolo del punto y coma con el que estamos finalizando todas las líneas de asignación de variables es una nueva adición a la sintaxis de «POV». En efecto, desde ahora, a partir de «POV 3.1», las sentencias de asignación a variables que asignen valores en flotante, vectores o colores de-

berán estar finalizadas por el símbolo del punto y coma, tal y como sucede con las sentencias de C. (Las declaraciones de objetos y texturas y las demás sentencias no acabarán en punto y coma). Si no ponemos este símbolo, «POV» comenzará a imprimir mensajes de warning (atención). Con todo, puede suceder que el fichero se compile y la escena se genere. Pero es muy probable que haya problemas, sobre todo si hay macros de por medio. Sólo por esta razón es muy probable que los escritores de pov-ipas deban adaptar su trabajo a la nueva versión.

Figura 2: Ejemplo de macro tomado de macro1.pov

```
        ...
        ...
// Definición de la macro Make_Many. Los tres parámetros que usa son...
// Stuff:  objeto que será copiado y rotado
// How_Many: Número de copias a hacer
// Axis:   El eje alrededor del cual han de rotar las copias

#macro Make_Many (Stuff, How_Many, Axis)
  #local Count=0;  // Este identificador es local y temporal a esta macro
  #while (Count<How_Many)
     object{Stuff rotate Axis*Count*(360/How_Many)}
     #local Count=Count+1;
  #end
#end    //aquí se acaba la definición de la macro


#declare Thing = cylinder{0,x,0.1}  //objeto cilíndrico

//aquí tenemos un ejemplo de uso de la macro
object{Make_Many (Thing,6,z)  // Hace 6 objetos Thing y los rota alrededor del eje Z
        pigment{Blue}
}
        ...
        ...
```

#Macro: ¿Funciones de verdad?

Si tenemos que escribir repetidamente un mismo trozo de código a lo largo de nuestro programa (o fuente, o fichero escénico), la tarea se nos hará tediosa y el propio texto del programa puede acabar haciéndose insufriblemente largo y fastidioso de leer. Sin embargo, ahora, a partir de la versión 3.1, podremos crear macros para ahorrar tiempo y conseguir programas más compactos y de más fácil lectura. Una macro es un conjunto de sentencias que se asignan a un nombre al que llamamos nombre de la macro. Una vez que una macro ha sido declarada, bastará con escribir su nombre en cualquier punto del programa para que «POV» inserte su código automáticamente en dicho punto, al compilar. A esto se le llama "expansión de la macro". Las macros tuvieron bastante importancia en la programación del ya vetusto lenguaje ensamblador pero en los lenguajes más modernos, donde existen cosas como funciones y objetos (que no tienen nada que ver con los objetos de «POV»), las macros apenas se usan.

Por supuesto, las macros convencionales no son demasiado flexibles pero afortunadamente las macros de «POV» no funcionan exactamente como las ya conocidas por los programadores. Las macros de «POV» tienen características que están a caballo entre las de las típicas macros y las de las funciones de C. El proceso de expansión de una macro de «POV» se realiza de manera idéntica al de una macro convencional pero los parámetros no funcionan igual. Echad un vistazo a la sintaxis de declaración de una macro, en la **Figura 1**. En esta estructura podemos distinguir tres partes. Por un lado está la cabecera de la macro, compuesta por la sentencia #macro, el nombre de la macro y los parámetros separados por comas. Luego está el llamado cuerpo de la macro, compuesto por sentencias del len-

Figura 1: Sintaxis de declaración de una macro

```
#macro nombre_macro(parametro 1, parametro 2, parametro n)
        sentencias
            ...
            ...
            ...
#end
```

Este pov-ipas de Joachin Puerschell es un de los muchos que existen para crear árb

Esta escena, obra de Nathan Kopp, utiliza los flares programados por el propio autor.

guaje escénico de «POV». Y finalmente tenemos la sentencia #end que marca el final de la macro. La macro puede utilizar parámetros o no, pero aunque no los tenga los paréntesis deberán escribirse. En cuanto al cuerpo de la macro, podemos emplear cualquiera de las sentencias de «POV». La única limitación está en que si utilizamos #if o #while, estas sentencias deberán estar completamente dentro (#end incluido) o completamente fuera de la macro.

Las macros tienen alcance global y si se emplea un mismo nombre para una nueva macro, la definición de la anterior se pierde. Las macros se emplean, simplemente escribiendo el nombre de las mismas en el lugar donde queremos que se inserte su código. Además del nombre debemos invocar a la macro escribiendo entre sus paréntesis tantos parámetros como se hayan empleado en la definición. Estos parámetros pueden ser valores en flotante, vectores, nombres de pigmentos, de texturas, de objetos, etc., y serán empleados por el código expandido por la macro. Echad un vistazo a la macro declarada e invocada en la **Figura 2**.

es.

Como podéis ver, la declaración de la macro utiliza tres parámetros: Stuff, How_Many y Axis. Fijaos también en la invocación a la macro. En ella se indican tres elementos que serán pasados a los parámetros. El primero, Thing, es un objeto de «POV» que hemos creado en una línea anterior, la segunda es un valor numérico y la tercera indica un eje para la rotación. Al encontrar esta invocación, «POV expande la macro y crea una tabla local de símbolos donde asigna los parámetros escritos en la invocación a variables locales que tienen el mismo nombre que hemos dado a dichos parámetros en la definición de la macro. Es en este punto donde las macros de «POV» se diferencian de las convencionales. Al expandirse una macro, durante la compilación, se crea siempre una tabla de símbolos locales con los parámetros. Esto, a todos los efectos, es lo mismo que si hubiéramos escrito las líneas siguientes al principio del cuerpo de la macro:

```
#local Thing = cylinder{0,x,0.1}
#local How_Many=6;
#local Axis=z
```

Si os fijáis, en el código de la macro os daréis cuenta del cometido de la misma. Como parámetros hemos dado a Make_Many un objeto –un cilindro– que es asignado al identificador Stuff, el cual es empleado dentro de la sentencia object de la macro para crear las copias. El número de éstas, indicado en el parámetro How_Many, es utilizado como límite por el contador del #while, con lo cual se crean seis copias. Y cada copia es rotada en torno al eje Axis (que vale z) un número de grados que depende del número de copias.

Finalmente, cuando «POV» halla el #end que corresponde a la macro, las variables locales de la macro, entre las que se incluyen las creadas automáticamente con los nombres de los parámetros, se destruyen. Es decir, cuando se sale del código de la macro, la línea que sigue a la de su invocación no sabe nada de las variables Stuff, How_Many o Axis, éstas no existen ya para el programa. Más tarde, si la macro vuelve a ser invocada, la tabla de símbolos se creará de nuevo y estas variables volverán a existir, aunque esta vez sus valores será probablemente distintos, dependiendo de lo que hayamos escrito para los parámetros. Es importante recordar un

par de cosas:

1) Al invocar a la macro hay que utilizar todos los parámetros que se han indicado en la declaración. De lo contrario habrá error.

2) Al emplear los parámetros, hay que asegurarse de que los tipos de datos pasados a los parámetros corresponden con los tipos definidos en la declaración. Cuando «POV» estudie la declaración de Make_Many, durante la compilación, comprobará que el identificador Stuff se utiliza como nombre de un objeto y por ello deberemos pasar un nombre de objeto a Stuff. Si, en lugar de esto le pasamos un número o cualquier otra cosa, «POV» nos dará un error.

Otro detalle importante es que cuando la macro crea un objeto (como en el ejemplo), este objeto no es local a la macro (con lo cual se destruiría al salir de la misma), sino que es devuelto por la macro de manera que puede ser asignado a un identificador y utilizado globalmente en el resto del programa. Por ejemplo, podríamos haber invocado a

```
#declare mio=object{Make_Many (Thing,6,z)}
```

la macro del ejemplo de este modo:

Esto crea el objeto mío.

Macros, variables locales y objetos y valores devueltos

Hemos visto cómo una macro puede ser empleada para devolver un objeto al programa. Pero la macro puede devolver también otras cosas. Si, por ejemplo, vamos a devolver un valor resultante de un cálculo, no necesitamos emplear #local ni #declare. En la **Figura 3** tenemos un ejemplo muy sencillo de esto tomado del fichero exper1.pov

Figura 3: Una macro que devuelve un valor.

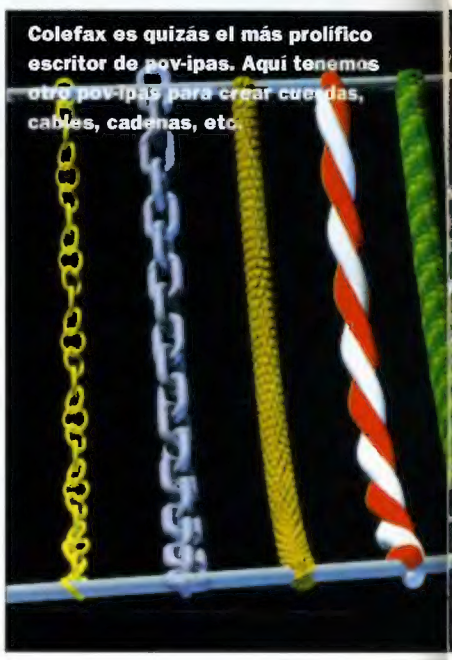
```
#macro duplica(kuk)
        kuk*2
#end

sphere{<0,0,0>,duplica(2) pigment{Red}}
        ...
        ...
```

(incluido en el CD-ROM).

En este caso la macro duplica devuelve un valor que es empleado por la sentencia sphere como radio para la esfera. (Damos el valor 2 como parámetro y se nos devuelve el valor 4).

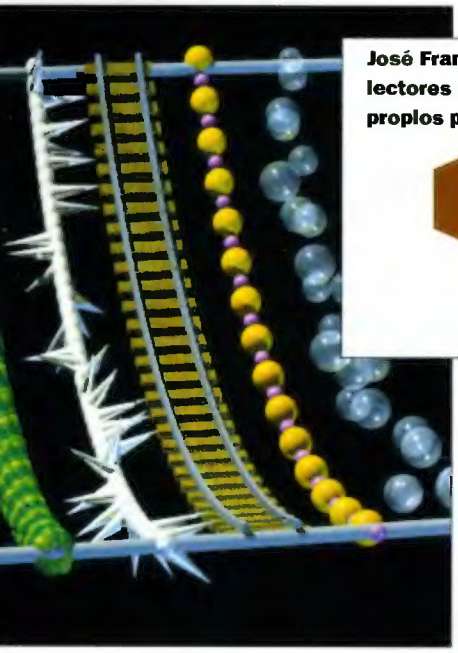
En fin, ahora que sabemos lo que es una macro podemos concluir la explicación anterior sobre el funcionamiento de variables locales. Supongamos que tenemos un proyecto formado por un fichero principal (.pov) que incluye a un fichero secundario (.inc) dentro del cual hay definida una macro. Recordad que «POV» considera que las variables locales lo son a las macros y a los ficheros, por tanto, en el presente ejemplo, «POV» entiende que hay tres ámbitos, el del fichero principal, el del secundario y el de la macro. En el caso de los ficheros, una variable local definida dentro de uno no puede ser conocida en otro archivo, pero el caso de las macros es distinto porque «POV» asume que el ámbito de la macro está incluido dentro del bloque del fichero del que forma parte. Bien, siguiendo con el ejemplo, en este proyecto hay definida una variable local llamada Turing en el fichero .inc, antes de la invocación a la macro, y hay otra variable local llamada Shannon definida dentro de la macro. Pues bien, la variable Turing podrá ser utilizada por la macro porque «POV» entiende que esta forma parte del bloque del fichero .inc. En



Colefax es quizás el más prolífico escritor de pov-ipas. Aquí tenemos otro pov-ipas para crear cuerdas, cables, cadenas, etc.

cambio, Shannon no será conocida por el fichero .inc porque es una variable local solamente a la macro. Como se ha explicado antes, Shannon será creada cuando la macro sea invocada (junto a otras variables para los parámetros, si los hay) y destruida cuando «POV» llegue al #end que cierra la macro. En cuanto al fichero principal, no podrá referenciar a Turing ni a Shannon.

Ahora vamos a considerar un par de casos de "choque" de variables. Se dice que hay choque de variables cuando utilizamos un mismo nombre para definir variables diferentes. Esto puede ocurrir en ciertos casos y no dará un error de sintaxis, aunque sí podemos cometer un error lógico si nos confundimos con la variable que estamos empleando. Suponed que hemos creado no una, sino tres variables llamadas Turing. La primera la definimos con #declare en el fichero principal dándole el valor 123, la segunda la definimos con #local en el fichero secundario otorgándole el valor 456, y la tercera la declaramos como local dentro de la macro asignándole el valor 789. Es



José Francisco García es uno de los pocos lectores que se ha animado a crear sus propios pov-ipas.

importante comprender que aquí no hay una, sino tres variables distintas que emplean el mismo nombre. «POV» ha reservado memoria para las tres y nuestro único problema será saber a cuál estamos referenciando en cada momento. Si escribimos la siguiente sentencia dentro de la macro (y des-

```
#local Turing=Turing+1;
```

pués de definir la Turing local):

¿Cuál de las tres variables estaremos empleando? Pues la de la macro, y esta pasará a valer 790. (Como regla para no perderos, debéis pensar que «POV» siempre emplea la variable local definida más recientemente, tal y como sucede en el lenguaje C). Después, al hallar el #end, «POV» destruirá la tabla de variables locales de la macro y la variable Turing de la macro será borrada de la memoria. Entonces estaremos nuevamente dentro del ámbito del fichero secundario

```
#local Turing=Turing+1;
```

y si volvemos a escribir:

estaremos alterando la variable local del fichero secundario, que quedará con el valor 457. Por último, cuando «POV» regrese al fichero principal (después de compilar todas las líneas del fichero secundario incluido), la tabla de variables locales del archivo secundario serán borradas. Únicamente, en adelante las nuevas modificaciones hechas a Turing se aplicarán sobre la variable global de este nombre.

Ahora volvamos al asunto de la devolución de valores por parte de las macros. Ya hemos visto cómo podemos hacer que una macro devuelva un valor. Hacer que modifique más valores tampoco es demasiado difícil. Simplemente bastará con escribir sentencias #declare para cambiar variables globales. Pero, ¿y si lo que deseamos es efectuar cambios en variables locales del fichero donde se ha expandido la macro?

Figura 4: Una macro que devuelve 2 valores en variables locales.

```
#macro alargaY(mio,b)
    #declare mio=object{mio scale<1,b,1>}
    #declare b=b*2;
#end
...
...
#local mio=sphere{<0,0,0>,2 pigment{Red}}
#local b=2;
alargaY(mio,b)
object{mio}
object{mio translate<7,0,0> pigment{Blue} scale<1,b,1>}
plane{y,0 pigment{White}}
...
...
```

En ese caso lo que debemos hacer es lo que podéis ver en la **Figura 4**, donde hemos incluido el fichero de prueba exper2.inc, el cual es cargado por el fichero principal exper2.pov (ambos se encuentran incluidos en el CD).

Como podéis, ver la macro alargaY acepta dos identificadores como parámetros; el identificador mio es un objeto, una esfera con dos unidades de radio y color rojo, y el identificador b es una variable que se utiliza para escalar en el eje Y al objeto pasado como parámetro. El resultado de este ejemplo es que se crea una esfera roja de radio 2 cuya longitud se multiplica por el valor de b (es decir, que se estira a lo largo de Y). Además, como el valor de b se devuelve multiplicado por 2, la siguiente esfera (de color azul) puede utilizar dicho valor para estirarse aun más. Observad que para devolver los cambios hechos en la macro a las variables locales, éstas han de modificarse utilizando #declare y no #local. Además, para que esto funcione son necesarias dos cosas. Primero que las variables se hayan creado como locales antes de invocar a la macro y, segundo, que utilicemos los identificadores de las variables locales como parámetros para la macro. Si empleamos valores constantes u objetos o cualquier otra cosa como parámetros (en vez de identificadores), las variables locales correspondientes no podrán ser alteradas por la macro. Si,

```
#local b=2;
alargaY(mio,2)
```

Esto no funcionará. La variable local b no será devuelta y, por tanto, el objeto azul mantendrá el mismo tamaño que el rojo. Además, si utilizamos como parámetro un identificador junto con una expresión (por simple que sea), esto tampoco funcionará. Este será el caso, por ejemplo. de:

```
alargaY(mio,b+1)
```

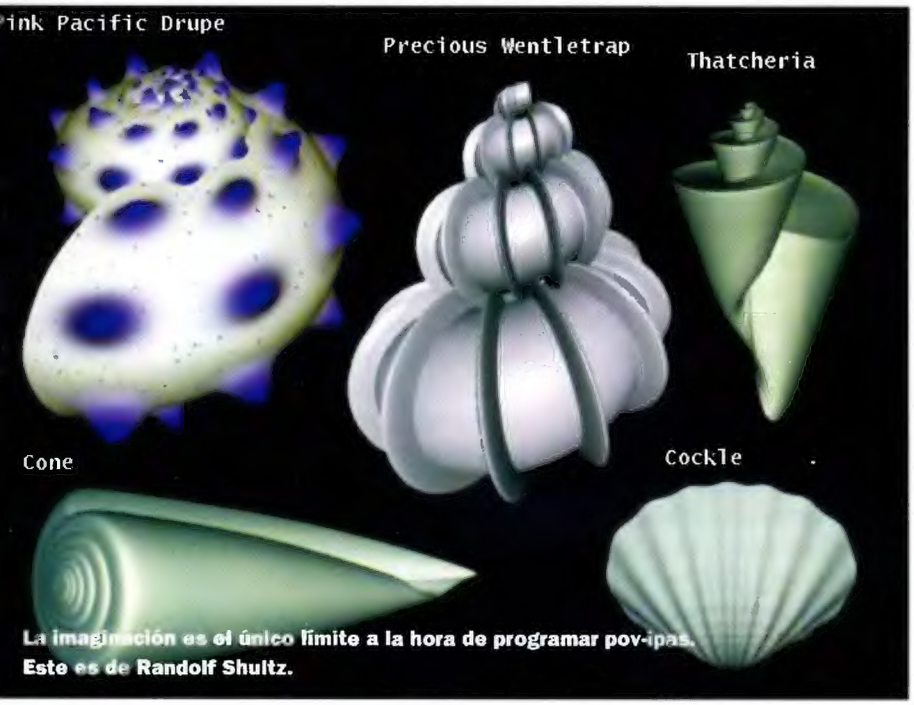
Para terminar, recordad que los valores devueltos de esta manera son locales. Ni mio ni b serán reconocidos desde el fichero principal.

Conclusiones

Todo esto puede parecer muy lioso y el lector puede sentir la tentación de evitar ciertos casos, como el uso de nombres repetidos con variables locales, e incluso puede optar por utilizar únicamente variables globales. Pero esto, a la larga, será un error. El propósito de las variables locales es precisamente el de impedir que haya demasiadas variables globales. Como éstas pueden ser alteradas en cualquier punto del programa, cuando se producen errores lógicos debemos rastrear su valor a lo largo de todo el listado. Además, es más fácil que se produzcan errores si hay un gran número de variables globales, ya que éstas pueden interactuar entre sí. Las variables locales, por el contrario, actúan como si estuviesen guardadas en compartimentos (los ficheros y las macros), con lo cual es mucho más improbable que interactuen de manera imprevista con otras variables. En general se considera que un listado donde predomine el uso de variables locales es mucho más fácil de leer. Por otro lado, el uso de variables locales hace que sea más fácil adaptar ficheros o macros de otros autores para nuestro propio uso. Finalmente, las variables locales sólo ocupan memoria mientras se está compilando el fichero o la macro a la que son locales y también se dice que su uso es más elegante. Por todas estas razones las variables globales sólo deben utilizarse cuando resulten imprescindibles, es decir, cuando se deba tratar una misma variable en distintos ficheros.

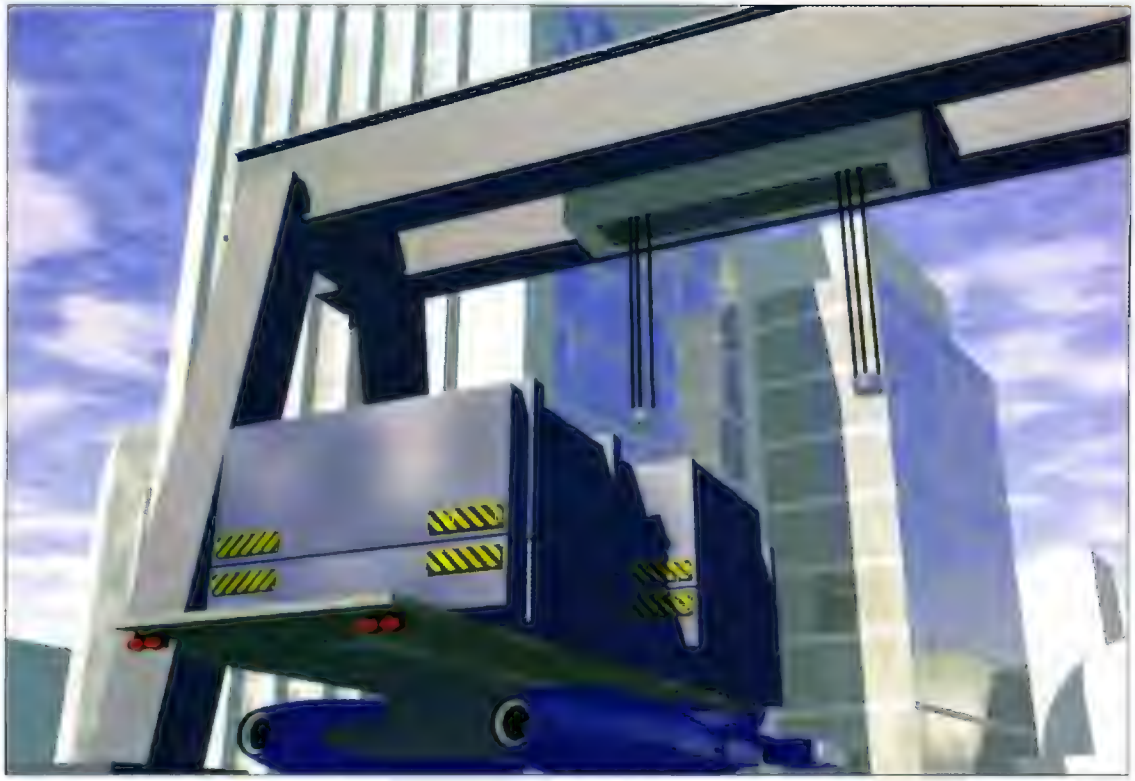
En cuanto al uso de un mismo nombre para distintas variables locales, esta situación puede darse en ciertos casos, como por ejemplo cuando necesitamos que haya recursividad. La recursividad se da en «POV» cuando hacemos que un fichero se incluya a sí mismo o una macro se referencie a sí misma. El código recursivo es más difícil de comprender pero supone una simplificación en ciertos casos.

En cuanto a las macros, podemos utilizarlas para simular funciones de C de manera mucho más sencilla y elegante que el método de inclusión de ficheros. Podemos crear macros que devuelvan valores como resultado del procesamiento de fórmulas complejas. Y también podemos definir macros que construyan objetos de uno u otro tipo en función de los parámetros con que se les invoque. Es una pena que las nuevas variables locales de «POV» no sean idénticas a las de C y que no haya prototipos para las macros, como sucede con las funciones de C, pero, aun así, #macro y #local nos permitirán crear buenos "programas escénicos". (En próximos números veremos ejemplos prácticos del uso de ambas sentencias).



La imaginación es el único límite a la hora de programar pov-ipas. Este es de Randolf Shultz.

Realizar animaciones con POV 3.1. (parte II)



Ya hemos comentado que la aparición de nuevas sentencias de programación de «POV 3.1» iba a suponer una pequeña revolución para los pov-usuarios. Pues bien, el campo de la animación no es ninguna excepción a ésto. Así que este mes, además de concluir con los aspectos básicos de la animación con «POV», estudiaremos la forma en que estas nuevas sentencias pueden servirnos para crear animaciones con nuestro raytracer favorito.

De nuevo conviene recordar que lo que vamos a explicar aquí, aparte de los aspectos básicos de la animación (que siguen siendo los mismos de siempre), no es la forma en que se anima con «POV». En lugar de esto, nos centraremos en la manera en que las últimas innovaciones de este programa pueden ayudarnos a crear animación. Por supuesto, esto quiere decir que todavía no existen ejemplos dignos de animaciones que utilicen arrays o macros, pero, con un poco de paciencia, nosotros mismos crearemos estos ejemplos en los próximos meses.

Obra hecha con «POV» para el concurso de animaciones da Internet.

Más comandos de animación

En el número anterior dijimos que la variable interna clock se incrementaba de manera proporcional al número de frames ordenados con los comandos +KFI y +KFF. Por defecto, clock se inicializa con valor 0 y se incrementa hasta llegar al valor 1 en el último frame de la animación, pero esto no tiene por qué ser obligatorio. Así, utilizando el comando de línea +KIn.n, podemos indicar un valor flotante n.n que será asignado a clock como valor inicial para el primer frame de la animación. Y, de igual forma, podemos indicar el valor final que ha de tener clock usando el comando de línea +KFn.n. Supongamos que invocamos a «POV» con los siguientes comandos de línea:

```
povray +iballs.pov +oballs.tga +w320 +h240 +KFI30 +KFF60 +KI2.5 +KF5
```

Esta línea creará una serie de archivos TGA que irán desde balls30.tga hasta balls60.tga. El valor inicial del reloj será de 2.5 para el primer frame a generar y de 5 para el último. Todos los demás valores de clock se calcularán de manera que cada frame tome un valor proporcional entre 2.5 y 5.

Ahora consideremos otra cuestión. Supongamos que hemos generado una animación de 500 frames con un valor de reloj que oscila entre 0 y 1. Hemos invocado a «POV» con la línea:

```
povray +iballs.pov +oballs.tga +w320 +h240 +KFI1 +KFF500
```

Sin embargo, después de contemplar los resultados, nos damos cuenta de que hay un fallo garrafal entre los frames 127 y 150. ¿Tal vez un objeto se ha superpuesto con otro? ¿La cámara ha pasado a través de un modelo con demasiada lentitud? No importa. Lo que sí importa es que tendremos que localizar el error, corregirlo, y volver a renderizar los frames erróneos. Por supuesto no queremos ni pensar en volver a renderizar todos los frames de la secuencia. ¿Cómo solventaremos, pues, el problema?

Probablemente estaréis pensando que basta con volver a invocar a «POV» con los comandos de línea +KFI127 +KFF150, pero esto sería un grave error. El problema estará en que haciendo esto, clock comenzará valiendo 0 para el frame 127 y acabará con el valor 1 para el frame 150, cuando, lógicamente, debería oscilar entre 0 y 1 para los 500 frames de la secuencia. Por supuest.o podríamos calcular el valor inicial y final que ha de tener clock para los frames de la lista 127-150, pero esto sería bastante fastidioso. Afortunadamente, «POV» nos ofrece otros comandos que nos sacarán del aprieto. Veamos, para arreglar el desaguisado, podríamos invocar a «POV» con la línea:

```
povray +iballs.pov +oballs.tga +w320 +h240 +KFI1 +KFF500 +SF127 +EF150
```

Con esta línea, «POV» entenderá que debe generar sólo el subconjunto de frames situado entre 127 y 150 del total de los 500 que forman la animación. Además, «POV» asignará a estos frames el valor de clock que les corresponde, con lo que el problema quedará resuelto. El comando SF indica el primer frame del subconjunto y EF indica el último. Lógicamente, no tenemos por qué recurrir a este truco sólo en caso de error. Podemos emplearlo para generar trozos determinados de animaciones, si sospechamos que puede haber algún problema con ellas o si deseamos realizar algún tipo de cambio.

Dos problemas sencillos de animación

A continuación estudiaremos dos casos sencillos para comprobar cómo debe emplearse clock.

Caso 1: tenemos dos modelos que representan a un par de brujos que van a emprender un combate de hechizos. En la primera parte de la animación deseamos que uno de ellos comience a crecer hasta hacerse tres veces mayor que su tamaño inicial.

Podemos resolver este caso de varias maneras. Si el único efecto que va a tener lugar durante esta secuencia va a ser el agigantamiento del modelo, lo más sencillo es usar los comandos KI y KF para poner el valor inicial del reloj a 1 y el valor final a 3. Además, tendremos que incluir esta sencilla transformación dentro del modelo del mago:

```
Object{mago1 scale<clock, clock, clock>}
```

Como veis, simplemente escribimos clock en los parámetros de scale para producir un escalamiento del modelo en todos los ejes del mismo. Como la variable vale 1 en el primer frame, el modelo quedará con el mismo tamaño en la primera escena a generar, y como el valor final es 3, el modelo tendrá un tamaño tres veces mayor que el inicial en el último frame de la animación. Naturalmente, para que esto funcione el modelo a escalar tendrá que tener los pies a la altura Y=0 y estar centrado con respecto a los ejes X y Z. Si este no es el caso, el modelo aumentará de tamaño pero también tendrá lugar un desplazamiento indeseado. (Nota para los pov-novatos: para comprender la razón de esto, considerad que el valor dado a scale en un eje multiplica todos los puntos del objeto a lo largo de dicho eje. Si la planta del pie está en Y=0, multiplicarla por 3 no tendrá ningún efecto. Si, en cambio, se halla en Y=2, después del escalamiento, su nueva posición estará en Y=6).

Naturalmente puede suceder que queramos realizar otros cambios en la animación, y por ello puede ocurrir que prefiramos dejar a clock en paz. En tal caso, si clock va a crecer desde 0 a 1, podemos emplear la siguiente fórmula:

```
Object{mago1 scale<1+(clock*2), 1+(clock*2), 1+(clock*2)>}
```

Caso 2: queremos realizar una animación de un sistema solar muy sencillo que, aparte del sol, sólo tiene un único planeta al que acompaña un satélite. Queremos que el planeta gire alrededor de su eje y que complete dos revoluciones en torno al sol, en una animación de 640 frames.

Para preparar correctamente esta animación tenemos que recordar que en «POV» las rotaciones siempre se efectúan en torno a los ejes de coordenadas, los cuales se cruzan entre sí en las coordenadas <0,0,0>. Esto quiere decir que si el planeta está situado en <0,0,0>, entonces una rotación en el eje Y le hará girar sobre sí mismo en torno a dicho eje. Si, en cambio, el planeta está, por ejemplo, en <100,0,0>, entonces la misma operación de rotación le hará dar vueltas en torno al punto <0,0,0>, pero no girará sobre sí mismo. (Nota: pedimos disculpas por explicar algo tan básico, pero si alguien no entiende esto le resultará complicado preparar animaciones con «POV». Si este es vuestro caso, será mejor que comencéis por leer el número 16 de Rendermanía). En la **Figura 1** tenemos un ejemplo de cómo podríamos crear el sistema solar deseado.

El caso del planeta es muy sencillo. Se supone que éste –al igual que el satélite– ha sido creado centrado en torno a los ejes de coordenadas. Por eso la primera operación que efectuaremos sobre él será la que le hará describir rotaciones en torno a su eje. Para ello hemos usado el valor 3600 para la rota-

Animación creada por VanSickle para el concurso de animaciones de Internet.

La animación de este castillo de juguete es obra de Nazrat Durand.

ción en Y, lo que significa que el planeta completará diez rotaciones en torno a su eje a lo largo de toda la animación. El siguiente paso es hacerle rotar en torno al sol, al que suponemos en <0,0,0>, y para ello simplemente trasladaremos al planeta a la distancia adecuada del sol (en el plano X-Z) antes de ordenar la siguiente rotación. Esta operación utiliza el valor 720, lo que significa que el cuerpo habrá completado dos revoluciones en torno al sol al llegar al fin de la animación.

En cuanto al satélite, su caso es algo distinto. Este cuerpo no gira en torno a sí mismo pero sí lo hace con respecto al planeta. Hemos definido arbitrariamente que la distancia entre ambos cuerpos es de 15 unidades. Por tanto, si el satélite ha sido creado centrado en <0,0,0>, nuestra primera tarea será colocarlo a 15 unidades del centro de coordenadas, a fin de poder preparar su rotación en torno al planeta. Esto se hace en la operación siguiente, en la que se ha empleado el valor 1800 para que el satélite complete cinco revoluciones en torno al planeta. Después aplicamos al satélite la misma traslación que al planeta y luego la misma rotación (para que siga a éste en su órbita en torno al sol).

Sin embargo, las distancias y las dimensiones utilizadas no tienen nada que ver con las que cabría esperar de los verdaderos cuerpos celestes. Más aún, las órbitas descritas son círculos perfectos en lugar de elipses, pero lo importante era exponer un caso de rotación tan simple como fuese posible.

Un aviso sobre las texturas

Existen algunas sentencias como jitter y crand que añaden a las texturas efectos que dependen de valores aleatorios. Estas sentencias pueden conseguir buenos resultados cuando lo único que queremos es crear una escena estática. Sin embargo, su uso debe evitarse a toda costa cuando se pretende crear una animación. Por ejemplo, en el caso de crand, esta sentencia intenta reproducir un efecto rugoso y granulado trabajando de manera aleatoria en los píxeles de la textura. Como dicho efecto es totalmente aleatorio (y trabajado pixel a pixel), el siguiente frame puede producir un efecto indeseado de ruido, ya que el "granulado" conseguido en la textura no tiene por qué ser igual en dos frames seguidos. (Nota: este problema también puede presentarse con cualquier programa que permita utilizar texturas procedurales y disponga de sentencias similares. Creemos que hace algún tiempo llegó a nuestras manos una animación de un lector –hecha con «Imagine»– y que tenía precisamente este problema).

Tim Latimer [illegible] esta carrera de coches de juguete.

El problema de las trayectorias

Antes de «POV 3.0» no existía ninguna forma –dentro del mismo fichero escénico– de decirle a «POV» que cambiase una acción por otra dependiendo del valor de clock. Está claro

Figura 1: trozo de código para generar el caso del planeta y su satélite.

```
...
...
object{planeta rotate y*3600*clock translate<100,0,0> rotate y*720}
object{satelite translate<15,0,0> rotate y*1800 translate<100,0,0> rotate y*720}
...
...
```

que ahora, gracias a #if, esto sí es posible. Ahora podemos tener un cuerpo moviéndose en una dirección durante un tiempo y ordenarle en un momento dado que se pare o que cambie de sentido o que haga cualquier otra cosa. Naturalmente, #switch también puede ayudarnos en estos casos y tenemos un sencillo ejemplo de su uso en la animación boing, donde #switch es utilizada para decidir qué fórmula debe aplicarse al movimiento de una pelota.

Sin embargo, con o sin sentencias de programación, aún tenemos que enfrentarnos a problemas bastante fastidiosos. Hasta ahora todos los casos que hemos expuesto han sido ejemplos de movimientos muy sencillos. Por supuesto podemos utilizar fórmulas más complejas para crear animaciones de cosas tales como caídas de cuerpos, rebotes de bolas de billar, etc. Asimismo también es muy fácil hacer materializaciones de objetos, alteraciones de escala, cambios de textura, movimientos de agua, etc. Pero todo esto no resuelve un problema típico con el que el usuario siempre debe enfrentarse: ¿cómo podemos definir una trayectoria compleja, llena de giros y cambios de sentido, para un cuerpo cualquiera?

La solución obvia es emplear la sentencia array. (Esta nueva característica de «POV» ya fue explicada en el número anterior, aunque los ejemplos de uso que se dieron eran casi todos de colocación de objetos. Ahora, en cambio, veremos cómo sacar partido a los arrays en el terreno de la animación). Una primera posibilidad es crear un array de una sola dimensión y guardar en él una serie de registros de movimiento para un cuerpo. Cada registro de este tipo podría estar organizado de la siguiente manera;

- campo 0: Vacío inicialmente.
- campo 1: Un valor en flotante que indica la posición X del próximo punto al que debe ir el cuerpo.
- campo 2: Idem para el eje Y.
- campo 3: Idem para el eje Z.

El array guardaría, pues, las posiciones espaciales por donde habría de pasar el cuerpo, lo que definiría su trayectoria. Ahora bien, ¿cómo nos las arreglaríamos para preparar la animación a partir de estos datos? Una posible solución podría pasar por hacer que el valor inicial de clock fuese igual a 1 y el valor final igual al número total de frames que queremos generar. Luego podríamos calcular la distancia total que ha de recorrer el cuerpo y dividirla entre el valor de clock para hallar la distancia que recorre el cuerpo para cada frame de animación. (Nota: estamos asumiendo que el cuerpo mantiene una velocidad constante durante toda la animación). Hecho esto no nos sería difícil calcular el valor de clock en que ha de terminar cada registro-desplazamiento (calculando la distancia entre la posición actual del cuerpo y la final indicada en el registro). Anotaríamos entonces el valor final de clock para cada registro de desplazamiento y luego, una vez ya en el bucle de animación, compararíamos el valor actual de clock con el almacenado en el registro en curso para determinar si continuamos tratando dicho registro o pasamos al siguiente. Para calcular el movimiento efectuaríamos incrementos en los ejes X, Y y Z. Incrementos que previamente habríamos calculado para cada registro al hallar la ecuación de la recta entre la posición final del registro anterior y la posición final del registro actual (o, en el caso del registro inicial, al hacer los cálculos entre la posición inicial del cuerpo y la final del primer registro). En cuanto al cálculo de la distancia entre puntos, emplearíamos la fórmula de la **Figura 2**.

(Nota: para no repetir los mismos cálculos iniciales en cada frame tendríamos que incluir a éstos en una comparación del tipo #if (clock=1)...). Por supuesto podríamos hacer que el array incluyese las trayectorias de movimiento de más cuerpos o bien utilizar más de un array para esto. Por tanto, lo único que tendríamos que hacer para crear una nueva animación sería poner clock con el valor total del número de frames deseados y escribir las posiciones 3D para las trayectorias.

Pero este método tampoco es perfecto y tiene dos problemas: el primero es que obliga a todos los cuerpos a mantener una velocidad constante y el segundo que las trayectorias de movimiento

Figura 2: Fórmula para hallar la distancia entre dos puntos.

Si tenemos dos puntos <x0, y0, z0> y <x1, y1, z1>, podemos definir la distancia entre los mismos con la formula:

$$\text{DISTANCIA} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

son siempre líneas rectas. Lo ideal sería poder definir cambios de velocidad y tener un procedimiento que interpretase los puntos 3D del array como puntos de una trayectoria spline, con lo que los movimientos estarían definidos por curvas suaves en lugar de por rectas. Aquí lo mejor sería crear una macro que pudiese ser utilizada por cualquier usuario, aunque éste no tuviese conocimientos matemáticos. De esta manera cualquiera podría definir trayectorias complejas para sus modelos fácilmente, aunque no supiese lo que es una curva spline. (Nota: nosotros somos un ejemplo perfecto de ignorancia e ineptitud en el campo de las mates, pero aun así, como la creación de una macro de este tipo nos parece un asunto de importancia capital, creemos que repasaremos algo sobre este tema).

Otro problema de gran importancia es el de la definición de movimientos para objetos antropomórficos. Sería ideal poder definir una matriz de movimientos para andar, correr, etc., y poder aplicarle leves variaciones aleatorias dentro de una animación. Si esto se logra, el crear una animación de un gran ejército en movimiento sería algo al alcance de la mano.

Una idea para un compilador de acciones

Otra idea que podría tener interés sería el de utilizar un array para guardar órdenes para los elementos que intervienen en las animaciones. De este modo, si ya hemos creado el código para cumplir esas órdenes, lo único que tendremos que hacer para preparar una animación cualquiera será preparar los modelos y cambiar los valores guardados en el array. Estos valores podrían formar registros similares al ya explicado antes, pero con la diferencia de que podría incluirse un campo de orden que indicase para qué son los restantes parámetros del registro. Así podrían definirse órdenes para la cámara, para las transformaciones en las texturas de los objetos, para sus desplazamientos, etc. De esta manera bastaría con escribir los valores del array para preparar una animación compleja, sin que tuviéramos que preocuparnos por escribir ningún código adicional. (Nota: la creación de un sistema como este implica bastantes quebraderos de cabeza, pero intentaremos crear un ejemplo de esto en un plazo tan corto como sea posible).

John Aughey ha creado esta animación sobre uno de los juguetes de Rubik.

Ideas para animaciones

Suponiendo que las ideas aquí expuestas no acaben resultando impracticables, ¿qué tipo de proyectos podríamos afrontar? Vamos a anotar las dos primeras ideas que han acudido a nuestra cabeza:

- Una partida de ajedrez. Podríamos utilizar un array para guardar registros con órdenes de desplazamiento para las piezas, con lo que podríamos crear nuevas animaciones (de otras partidas) muy fácilmente, con sólo cambiar la partida "escrita" en el array. Además, podríamos crear código para que cada pieza se "comiese" a las demás de una manera distinta (como en Killer Chess y otros programas parecidos).
- ¡Una batalla de tanques en una ciudad! De nuevo podríamos utilizar los arrays para guardar valores de trayectorias a fin de definir las evoluciones de los carros de combate. Si creamos los edificios de la ciudad desde un modelador gráfico sería sencillo tomar nota de las posiciones 3D que nos interesasen, a fin de definir las evoluciones de los tanques.

La animación de un ejército medieval es algo en lo que por el momento será mejor no pensar. También podría ser muy interesante el escribir un programa de verdad, con comportamientos para los personajes, de manera que ellos mismos pudiesen definir sus propios movimientos. A partir de «POV 3.1», esto es ya teóricamente posible, aunque de momento será mejor pensar en cosas mas sencillas.

Nota importante. Podéis remitirnos vuestros trabajos o consultas, bien por carta a la dirección que figura en el sumario de Pcmanía, o vía e-mail a rendermania.pcmania@hobbypress.es

Animaciones a lo grande (Parte II)

Continuamos publicando animaciones de gran extensión remitidas por los lectores. Queremos recalcar que el foro de este mes es una excepción a la regla del tamaño, algo que sólo ha sido posible por circunstancias especiales. Así, aprovechamos para advertiros que no enviéis trabajos superiores a los 5 Megas comprimidos.

Nuestro amigo **Víctor Barbero Romero** nos ha enviado unas espectaculares escenas renderizadas con «POV 3.0» de un magnífico piano modelado con «Spatch» y de un encuentro donde un caballero pilotando un caza da una paliza a dos pobres orcos. Bien por tí, pero parece un poco fuera de lugar que suministres un caza para el bando humano en un mundo de clara ambientación y tecnología medievales. En cuanto a tus cartas, parece que tienes una pequeña confusión con las medidas de nuestros modelos. En las escenas los orcos y caballeros tienen una altura aproximada de entre 160 a 190 unidades porque en ellas 100 unidades equivalen a un metro.

Aunque no te lo creas, la versión de «POV» de Windows no es más rápida que la de MS-DOS (bueno, en un render de una hora y viendo la imagen mientras se lanza, la versión de Windows puede ser unos segundos más lenta). Lo que sucede es que sí requiere más memoria (por algo funciona bajo Windows). Esto puede hacer que, si tienes poca memoria, el programa tenga que hacer muchos más accesos a disco duro de los que precisaría bajo MS-DOS, con lo cual el render sí será mucho más lento. Por otro lado, si estás renderizando y pones la ventana de «POV» para Windows en un segundo plano, el programa también irá más lento . Y para terminar, ¿qué prioridad has puesto para la velocidad del render? (Mírate la opción "Render\ render priority" y ponla a hightest). En cuanto a tu último coemntario de tu carta tenemos que decir que sentimos lo de tu versión de «Rhino». La verdad es que salió una nueva beta de libre uso después de la tuya, pero PCmanía decidió no publicarla porque sólo iba a tener un par de meses de vigencia. Estaremos atentos a este asunto por si aparecen nuevas versiones.

Gracias por tus ánimos y toma nota de nuestras felicitaciones por tu habilidad con «POV» y «Spatch» (y sobre todo por el piano y sus texturas).

El Foro del Lector

A pesar de creer que hemos visto antes la metralleta medieval (creada con el programa «POV») de **Kepa Diez Ara**, no estamos seguros de haberla publicado, así que incluimos aquí estas escenas (¿las mismas?) enviadas con tu nueva carta. Tu metralleta será utilizada en la próxima batalla (¿quizá contra las hambrientas evormigas?). Y aquí va la contestación a algunas de tus preguntas:

Con respecto a «Imagine», no conocemos otro "traductor" de modelos aparte de 3dto3d y 3Dwin.

POr tora parte, solemos dejar sin tocar el assumed_gamma que viene por defecto. Preferimos trastear con la intensidad y colocación de las luces. En cuanto a tu quinta pregunta, emplearíamos «Spatch» y «POV» o «Rhino» y «POV» o cualquier otro modelador con «POV».

Gracias por tu metralleta y enhorabuena por tu horrorosa transformación (seguimos esperando ese mech hecho con el programa «Imagine»).

Antonio Delgado Castillo es otra nueva víctima del virus infográfico y pide algunos consejos sobre cómo empezar con «MAX». Bien, lo que sigue no te va a parecer demasiado original pero lo que necesitas es practicar, practicar y (además) practicar. Para empezar no intentes realizar proyectos demasiado complejos. Comienza intentando hacer modelos sencillos, como piezas de ajedrez y objetos por el estilo, y de momento olvídate de cosas como el modelado orgánico. No podemos enviarte manuales y, en cuanto a libros, quizá el mejor sea el de Roberto Potenciano. De todos modos no te limites a seguir mecánicamente los ejercicios descritos en los libros e idea tus propios proyectos (al tiempo que estudias cómo llevarlos a cabo). Los comienzos pueden ser duros, pero si te aplicas acabarás disfrutando a lo grande.

Roberto Ajenjo Murcia es otro povmaníaco recién enganchado que nos envía este mes sus primeras obras. ¿Una crítica? Bueno, tus escenas son prometedoras pero aún tienes que practicar bastante con el modelado y la creación de texturas. En cuanto a tus preguntas, algunas de ellas ya han sido contestadas en los últimos números, y respecto a las demás:

3) ¿Números atrasados de Rendermanía sin comprar PCmanía? Nos tememos que esto no está previsto.

4) Llama a la redacción y pide que comprueben el estado de tu pedido.

5) Lo de los plug-ins de «MAX» tiene bastante miga y además, por ahora, no podemos hablar de ello aquí.

6) Sobre «POV», aquí sólo se ha publicado el "Raytracing II" de Anaya, pero este libro está ya bastante desfasado, pues trata la versión 2.2 de «POV» (cuando el programa ya va por la 3.1). De todas formas, para cuando leas esto, los últimos números deberían haberte ayudado bastante con nuestro raytracer favorito.

Aunque últimamente es frecuente recibir e-mails de Argentina no lo es tanto el recibir cartas de ese país por vía postal. Este ha sido el caso de la misiva de **Jesús Nicolás Martín Colom**, en la que este autor nos envía unas –realmente– fantásticas imágenes creadas con «POV», «Imagine» y «3D Studio». Entre ellas hay un par de magníficos mechs, un increíble helicóptero y un jeep. Con todo hay que decir que en algunas imágenes, como en imagine-7 o imagine-9, parecen retocadas a mano y a veces se hace difícil apreciar debidamente el modelo. En cuanto a tus preguntas:

1) Como sabes, para simular una reflexión sobre un objeto desde «3D Studio», se indica el nombre de una textura en el indicador reflection del cuadro del Materials Editor, pero creemos que no te refieres a esto, sino a las llamadas reflexiones automáticas. Para conseguirlas debes pulsar sobre el recuadro de "A", en el recuadro map de reflection. (No somos unos expertos en «3D Studio» pero creemos que se trata de esto).

2) No acabamos de entender lo que dices del "cono de luz" en la atmósfera. «3D Studio» no puede producir directamente estos efectos, aunque quizá sea posible simularlos aplicando una textura procedural con transparencia parcial sobre un objeto en forma de cono.

3) Creemos que los lpas que generan luces flare en «3D Studio» funcionan como lo hacen los flares de «Polyray», «POV» y otros programas. Colocan una textura sobre un objeto bidimensional. Suponemos que podrías hacer lo mismo manualmente, creando una textura adecuada.

Quienes recuerden la excelente animación que publicamos de este autor en el número 7 de Rendermanía, sabrán que **Luis Cárdenas Salido** es un infografista con un gran interés en los efectos. Así, en las dos nuevas animaciones que Luis nos ha remitido veremos efectos de lluvia, viento, de luz y de partículas. También hay algún que otro morphing, relámpagos, y otras cosas. Amigo Luis, sentimos haber tardado tanto tiempo en publicarte esto, pero es que no nos es fácil publicar trabajos de tanta extensión. Enhorabuena por tus efectos.

Fernando Herrera Rodríguez (Mixerable Productions) nos envía una animación taurina que ha dado algunos problemas (hemos tenido que ver los flis uno a uno). Como Fernando nos pide una crítica, aquí la añadimos, a pesar de que él mismo se hace una propia. Para empezar la animación de los personajes es demasiado rígida y la propia animación del toro no se corresponde con la que debería tener este animal, recordando por el contrario a la de un insecto. Además, los vuelos de cámara son excesivamente rápidos e impiden captar los detalles de las películas. En fin, te ha fallado un poco la planificación del trabajo. Y de tus preguntas...

1) Los objetos construidos desde «POV» no tienen malla así que no pueden ser exportados fácilmente a «3D Studio». Que sepamos nadie ha hecho ninguna herramienta capaz de hacer esto.

2) Uno de los mejores sitios para encontrar buenos modelos de libre uso es 3Dcafe en www.3dcafe.com.

3) «MAX» tiene una opción para exportar sus objetos a «3D Studio», aunque se pierde calidad debido al problema de la exportación de texturas.

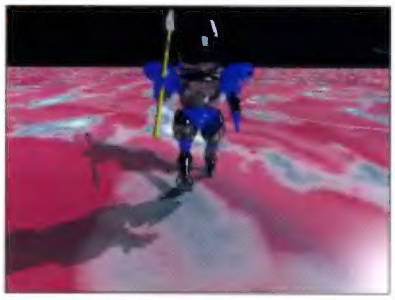
4) No podemos publicar librerías completas de modelos sin el consentimiento de los autores de las mismas.

5) Pasamos tus cuestiones sobre Pixel a Pixel a los responsables del concurso.

El Foro del Lector

Los hermanos **Bruno** y **Alberto Otero Leal** nos envían esta animación que publicamos a pesar de que sólo en parte está compuesta por imagen sintética. Anotaros un tirón de orejas por no incluir ninguna carta para el foro del lector con los detalles de vuestro trabajo.

Otra carta rescatada del olvido es la de **Carlos Pastor Méndez** que nos envió esta misiva a finales del año pasado. En ella Carlos pone nuevas texturas al antiguo lancero de Rendermanía y adjunta algunas escenas creadas con «POV». En cuanto a las preguntas de tu carta (la escrita a mano), preferimos que enviéis las escenas en formato JPG ya que es quizá el que más comprime sin apenas pérdida de calidad. Y sobre los modeladores, bien, si todavía continuas leyendo este suplemento ya habrás leído sobre programas como «Spatch» y «Rhinoceros» y es probable que dentro de poco presentemos otros nuevos. Perdona por el retraso.



Javier Arruabarrena Larequi nos envía una animación realizada con «MAX» sobre el grupo Los Suaves. La animación incluye música del grupo e imágenes escaneadas de las portadas de sus discos además de algunas notas relacionadas con dicho grupo. Gracias por tu felicitación, esperamos tus próximos trabajos.

Alberto Sendín Domínguez, el famoso autor de las aventuras virtuales de Piolín (y Mazinger), vuelve de nuevo a estas paginas con una animación de... ¡54 Megas comprimidas! Este tamaño ya de por sí hubiera hecho muy difícil su publicación, pero es que, además, en el momento de escribir estas líneas no disponíamos de ninguna unidad ZIP, por lo que nos ha sido imposible ver tu trabajo. Con todo, como ya te conocemos, estamos seguro de que valdrá la pena publicarte, así que hemos pasado tu ZIP a los responsables de la maquetación que sí poseen esta unidad de ZIP). Así que tendremos que esperar a que aparezca el número correspondiente para ver tu animación, como todos los demás. Mientras tanto remitimos a los lectores a tu carta en el foro.